Recursion

CSC 209 Data Structures

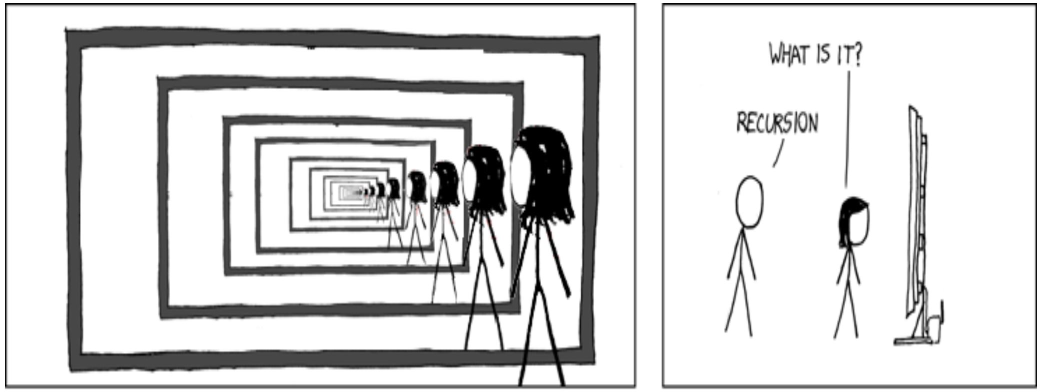
Kulwadee Somboonviwat

kulwadee.som [at] sit.kmutt.ac.th

Adapted from: lecture slides of Robert Sedgewick.

Overview

- What is recursion?
  - When something is specified in terms of itself.



http://xkcdsw.com/content/img/1105.gif

- Why learn recursion?
  - Represent a new mode of thinking.
  - Provides a powerful programming paradigm.
  - Many computational artifacts are recursive (i.e. self-referential) – trees, file systems, divide-and-conquer

Example: Digits -> Binary

- **Recursive program**

  To compute a function of a positive integer *N*

  - **Base case**: return a value for small *N.*
  - **Reduction step**: Assuming that it works for smaller values of its argument, use the function to compute a return value for *N*.

```java
public class Binary {
    public static String convert(int N) {
        if (N == 0) return "0";
        if (N == 1) return "1";
        return convert(N/2) + (N % 2);
    }
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        if (N < 0) {
            System.out.println("N must be positive.");
        } else {
            System.out.println(convert(N));
        }

    }
}
```

How can we be convinced that our recursive method is correct?
→ Use *mathematical induction.*

Mathematical Induction

To prove a statement involving a positive integer *N*

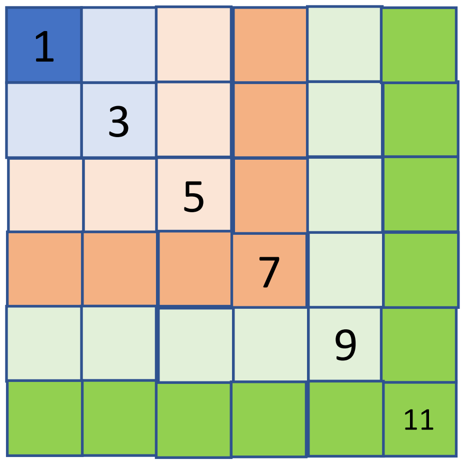
- **Base case**. Prove it for some specific values of *N*.
- **Induction step**. Assuming that the statement is true for all positive integers less than *N*, use that fact to prove it for *N*.

Example. The sum of the first *N* odd integers is N^2 .

Base case. True for $N=1$.

Induction step. The *Nth* odd integer is $2N - 1$.
Let $T_N = 1 + 3 + \dots + (2N - 1)$ be the sum of the first *N* odd integers.

- Assume that $T_N = (N-1)^2$,
- Then, $T_N = (N-1)^2 + (2N - 1) = N^2 - 2N + 1 + 2N - 1 = N^2$



Proving a recursive program correct

Recursive program

To compute a function of a positive integer *N*

- **Base case**: return a value for small *N*.
- **Reduction step**: Assuming that it works for smaller values of its argument, use the function to compute a return value for *N*.

Mathematical Induction

To prove a statement involving a positive integer *N*

- **Base case**. Prove it for some specific values of *N*.
- **Induction step**. Assuming that the statement is true for all positive integers less than *N*, use that fact to prove it for *N*.

```
public static String convert(int N) {
    if (N == 0) return "0";
    if (N == 1) return "1";
    return convert(N/2) + (N % 2);
}
```

convert*(N)* computes the binary representation of *N*

- **Base case**. Return "0" for *N*=0 and "1" for *N*=1.
- **Induction step**. Assuming that convert*(N/2)* is correct.
  1. if *N* is even (*N % 2* == 0), then "0" is append to the result. Thus, convert(*N*) gives a correct result.
  2. if *N* is odd (*N % 2* == 1), then "1" is append to the result. Thus, convert(*N*) gives a correct result.

Mechanics of a function call

System actions when any function is called

- *Save environment* (values of all variables and call location).
- *Initialize values* of argument variables
- *Transfer control* to the function.
- *Restore environment* (and assign return value)
- *Transfer control* back to the calling code.

```java
public class Binary {
    public static String convert(int N) {
        if (N == 0) return "0";
        if (N == 1) return "1";
        return convert(N/2) + (N % 2);
    }
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        if (N < 0) {
            System.out.println("N must be positive.");
        } else {
            System.out.println(convert(N));
        }

    }
}
```

```
convert(26)
   if (N == 0) return “0”;
   if (N == 1) return “1”;
   return convert(13) + “0”;
convert(13)
   if (N == 0) return “0”;
   if (N == 1) return “1”;
   return convert(6) + “1”;
convert(6)
   if (N == 0) return “0”;
   if (N == 1) return “1”;
   return convert(3) + “0”;
convert(3)
   if (N == 0) return “0”;
   if (N == 1) return “1”;
   return convert(1) + “1”;
convert(1)
   if (N == 0) return “0”;
   if (N == 1) return “1”;
   return convert(3) + “0”;
```

Mechanics of a function call

System actions when any function is called

- *Save environment* (values of all variables and call location).
- *Initialize values* of argument variables
- *Transfer control* to the function.
- *Restore environment* (and assign return value)
- *Transfer control* back to the calling code.

```
public class Binary {
    public static String convert(int N) {
        if (N == 0) return "0";
        if (N == 1) return "1";
        return convert(N/2) + (N % 2);
    }
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        if (N < 0) {
            System.out.println("N must be positive.");
        } else {
            System.out.println(convert(N));
        }

    }
}
```

```
convert(26)
   if (N == 0) return "0";
   if (N == 1) return "1";
   return convert(13) + "0";
convert(13)
   if (N == 0) return "0";
   if (N == 1) return "1";
   return convert(6) + "1";
convert(6)
   if (N == 0) return "0";
   if (N == 1) return "1";
   return convert(3) + "0";
convert(3)
   if (N == 0) return "0";
   if (N == 1) return "1";
   return "1" + "1";
```

Mechanics of a function call

System actions when any function is called

- *Save environment* (values of all variables and call location).
- *Initialize values* of argument variables
- *Transfer control* to the function.
- *Restore environment* (and assign return value)
- *Transfer control* back to the calling code.

```
public class Binary {
    public static String convert(int N) {
        if (N == 0) return "0";
        if (N == 1) return "1";
        return convert(N/2) + (N % 2);
    }
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        if (N < 0) {
            System.out.println("N must be positive.");
        } else {
            System.out.println(convert(N));
        }

    }
}
```

```
convert(26)
    if (N == 0) return "0";
    if (N == 1) return "1";
    return convert(13) + "0";
convert(13)
    if (N == 0) return "0";
    if (N == 1) return "1";
    return convert(6) + "1";
convert(6)
    if (N == 0) return "0";
    if (N == 1) return "1";
    return "11" + "0";
```

Mechanics of a function call

System actions when any function is called

- *Save environment* (values of all variables and call location).
- *Initialize values* of argument variables
- *Transfer control* to the function.
- *Restore environment* (and assign return value)
- *Transfer control* back to the calling code.

```
public class Binary {
    public static String convert(int N) {
        if (N == 0) return "0";
        if (N == 1) return "1";
        return convert(N/2) + (N % 2);
    }
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        if (N < 0) {
            System.out.println("N must be positive.");
        } else {
            System.out.println(convert(N));
        }

    }
}
```

```
convert(26)
   if (N == 0) return "0";
   if (N == 1) return "1";
   return convert(13) + "0";
```

```
convert(13)
   if (N == 0) return "0";
   if (N == 1) return "1";
   return "110"+ "1";
```

Mechanics of a function call

```
convert(26)
    if (N == 0) return "0";
    if (N == 1) return "1";
    return "1101" + "0";
```

System actions when any function is called

- *Save environment* (values of all variables and call location).
- *Initialize values* of argument variables
- *Transfer control* to the function.
- *Restore environment* (and assign return value)
- *Transfer control* back to the calling code.

```java
public class Binary {
    public static String convert(int N) {
        if (N == 0) return "0";
        if (N == 1) return "1";
        return convert(N/2) + (N % 2);
    }
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        if (N < 0) {
            System.out.println("N must be positive.");
        } else {
            System.out.println(convert(N));
        }

    }
}
```

Mechanics of a function call

System actions when any function is called

- *Save environment* (values of all variables and call location).
- *Initialize values* of argument variables
- *Transfer control* to the function.
- *Restore environment* (and assign return value)
- *Transfer control* back to the calling code.

```
convert(26)
   if (N == 0) return "0";
   if (N == 1) return "1";
   return "1101" + "0";
```

```java
public class Binary {
    public static String convert(int N) {
        if (N == 0) return "0";
        if (N == 1) return "1";
        return convert(N/2) + (N % 2);
    }
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        if (N < 0) {
            System.out.println("N must be positive.");
        } else {
            System.out.println(convert(N));
        }

    }
}
```

"11010"

```
kulwadee-mbair:lect05 ann$ java Binary 26
11010
```

Greatest Common Divisor

- The largest integer that evenly divides into two integers p and q.
- Ex

  gcd(192, 24) = 24

  gcd(1025, 75) = 25

- **Applications**
  - Simplify fractions: 75 / 1025 = 3 / 41
  - RSA cryptosystem.

Greatest Common Divisor

- The largest integer that evenly divides into two integers p and q.
- Euclid's algorithm. [Euclid 300 BCE]

$$\gcd(p,q) = \begin{cases} p & \text{if } q = 0 \\ \gcd(q,\, p \,\%\, q) & \text{otherwise} \end{cases}$$

← base case

← reduction step, converges to base case

Java implementation.

```
public static int gcd(int p, int q) {
    if (q == 0) return p;
    else return gcd(q, p % q);
}
```

← base case

← reduction step

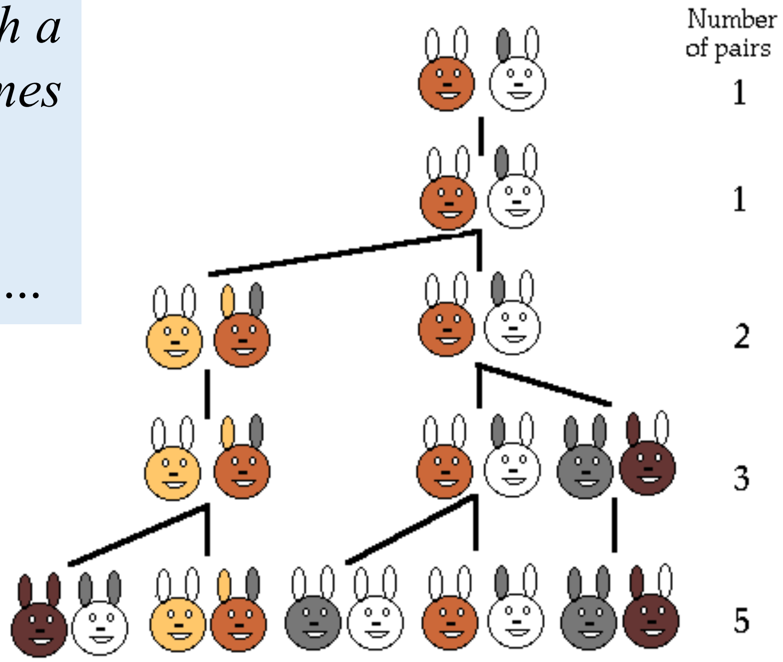
Fibonacci Numbers

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

Leonardo Pisano Bogollo
(1170-1250), Italy

"How many pairs of rabbits will be produced in a year, beginning with a single pair, if in every month each pair bears a new pair which becomes productive from the second month on?"

The result can be expressed numerically as: 1, 1, 2, 3, 5, 8, 13, 21, 34 ...



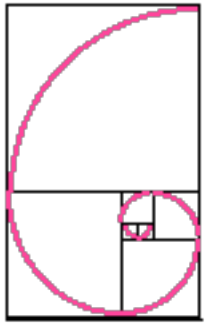
http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibnat.html

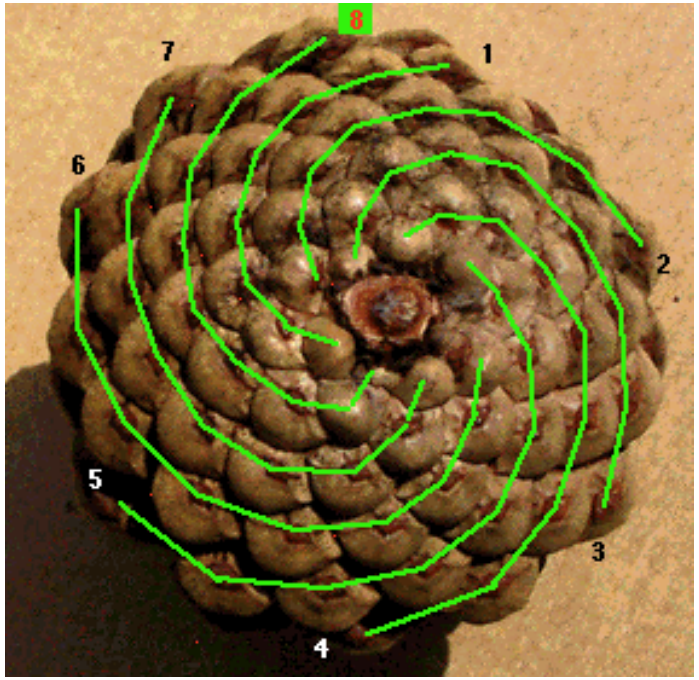
Fibonacci Numbers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

Leonardo Pisano Bogollo
(1170-1250), Italy





Fibonacci Numbers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

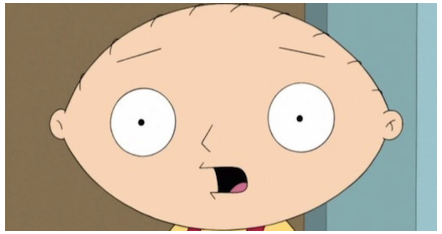
Leonardo Pisano Bogollo
(1170-1250), Italy

```java
public static long F(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return F(n-1) + F(n-2);
}
```

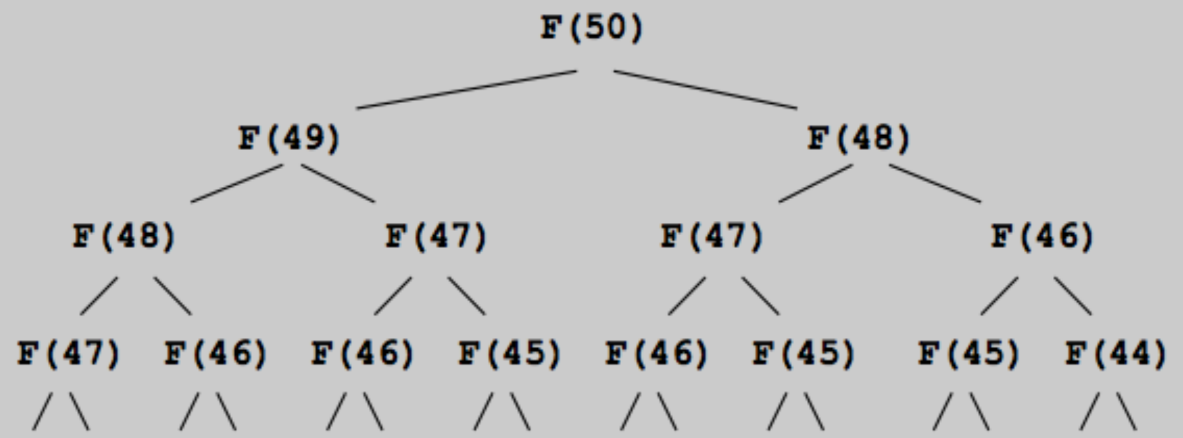
Is this efficient ?

```
public static long F(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return F(n-1) + F(n-2);
}
```

Is this efficient ?



NO! It is horribly inefficient!



recursion tree for naïve Fibonacci function

F(50) is called once.

F(49) is called once.

F(48) is called 2 times.

F(47) is called 3 times.

F(46) is called 5 times.

F(45) is called 8 times.

...

F(1) is called 12,586,269,025 times.

F(50)

An efficient way to compute Fibonacci Numbers

```
public static long F(int n) {
    if (n == 0) return 0;
    long[] F = new long[n+1];
    F[0] = 0;
    F[1] = 1;
    for (int i = 2; i <= n; i++)
        F[i] = F[i-1] + F[i-2];
    return F[n];
}
```

- <u>Idea</u>: use an array to record previously computed numbers. No Recursion!
- This is an example of a programming technique called **dynamic programming**

Rules of thumb in developing recursive programs

- Base case – always include a conditional statement as the first statement in the program that has a return
- Recursive calls must converge to the base case
- Recursive calls should not address subproblems that overlap

Vocabulary

- **iterative** : a method or algorithm that repeats steps using one or more loops.

- **recursive**: a method or algorithm that invokes itself one or more times with different arguments.

- **base case**: a condition that causes a recursive method *not* to make another recursive call.

- **factorial**: the product of all the integers up to and including a given integer.

Ref: https://books.trinket.io/thinkjava2/chapter8.html

Exercise

- Write a *recursive* program that removes all the character ‘x’ from an input string.

Exercise

- Write a *recursive* program that removes all the character ‘x’ from an input string.

```
public static String noX(String s) {
    if (s.length() == 0) return "";
    if (Character.toLowerCase(s.charAt(0)) == 'x') {
        return noX(s.substring(1));
    } else {
        return s.charAt(0) + noX(s.substring(1));
    }
}
```

Exercise

- Write a *recursive* program that counts the number of times that the value *17* appears in an input array of integers.

Exercise

- Write a *recursive* program that counts the number of times that the value *17* appears in an input array of integers.

```
public static int count17(int[] arr, int index) {
    if (index >= arr.length) return 0;

    if (arr[index] == 17) {
        return 1 + count17(arr, index+1);
    } else {
        return 0 + count17(arr, index+1);
    }
}
```